AIAA-98-4529

LaSRS++ AN OBJECT-ORIENTED FRAMEWORK FOR REAL-TIME SIMULATION OF AIRCRAFT

Richard A. Leslie, David W. Geyer, Kevin Cunningham*
Patricia C. Glaab, P. Sean Kenney, Michael M. Madden*

Unisys Corporation
NASA Langley Research Center
MS 169
Hampton, VA 23681

Abstract

Frameworks represent a collection of classes that provide a set of services for a particular domain; a framework exports a number of individual classes and mechanisms which clients can use or adapt. This paper presents an overview of an object-oriented (OO) framework that can be used both in an interactive mode from a desktop, or to support hard, synchronous, pilot-in-the-loop real-time aircraft simulations. The abstractions used and the benefits of object technology in this environment will be discussed. The framework described in this paper has been adopted at NASA Langley Research Center (LaRC) as the basis for all future real-time aircraft simulations.

Introduction

The decision to shift the real-time simulation environment from procedural FORTRAN to OO C++ was made after serious consideration. To provide insight into this decision a brief history of the evolution of real-time simulation at LaRC over the last eight years is in order.

The legacy real-time simulation environment at LaRC was procedural FORTRAN and involved the use of several (more than six) separate software repositories. Each repository was optimized to support a specific type of research using a different simulator cockpit. Repositories were independent of each other with different variable naming conventions, `COMMON` block structures, or functional breakdown.

The fact that each repository was its own separate development environment made it difficult to move personnel from project to project due to the learning curve involved. Few (if any) developers fully understood all of the repositories. It was difficult to balance the workload across a limited development staff as the number of airplanes active in each repository fluctuated. Moving code from repository to repository was difficult. Significant modification and testing were required to move an aircraft model to a different repository. In the past it was felt that these costs were acceptable as development was being done on hardware that had limited memory and processing power.

In the early 1990s LaRC began to invest in multiprocessor hardware platforms with relatively large amounts of memory and computing power. As these machines became operational, discussions began about consolidating repositories in order to have a more unified environment. This resulted in a project to consolidate the FORTRAN repositories. The resulting repository was capable of meeting LaRC simulation requirements with one exception, it could not support multi-vehicle simulations. Multi-vehicle work was supported by a separate repository that used dimensioned state variables to allow up to three vehicles to be simulated simultaneously. This repository supported distributing the simulation over two CPUs. The decision to maintain a separate repository for multi-vehicle work was due to a reluctance to impose the complexity of dealing with dimensioned state variables on single vehicle projects.

* Senior Member, AIAA

In May 1994, a prototyping effort was started to explore the feasibility of using object technology to develop a single framework that would be capable of supporting all real-time aircraft simulation at LaRC. This prototype eventually evolved into the Langley Standard Real-Time Simulation in C++ (LaSRS++) framework. The decision to make LaSRS++ the standard framework was made February 1995.

Key OO Concepts Utilized In Framework

Four basic OO concepts utilized in the LaSRS++ framework are:

1. Abstraction
2. Encapsulation/Containment
3. Inheritance
4. Polymorphism

The ability to deal with layers of complexity is facilitated by the use of abstraction. The process of abstracting away complexity allows the framework developer to provide the user with a minimal but complete interface that, through abstraction, encapsulates the details of an object's behavior. While being developed, the particular details of an object's behavior must be managed. Deciding the appropriate level of abstraction is a design challenge. In developing classes for the framework the decision was made to initially provide the minimal interface to a component. The justification being that it is easier to add methods to access encapsulated data/behavior than it is to reduce an over populated interface. The provided interface to a component constitutes a contract with the users, adding a method does not affect the current users, removing a method is a violation of the implicit contract. Violation of the contract forces the user to make modification to source code that uses the modified class. If the contract remains intact, and just the internal implementation changes, then the user can recompile and relink without having to modify client source code.

The C++ language provides a means to control a users access to the implementation details and data in a class. In the LaSRS++ framework, the decision was made to make all data private. Public and protected accessors are provided to allow the users and derived classes, respectively, to access to the data. Data and methods that are implementation details are not made public thereby protecting the integrity of the data and retaining a level of control of an objects state. This prevents accidental corruption of data contained in the object. Furthermore, access through member functions allows the function that returns a value to be replaced by one that executes an algorithm and returns the result without users having to modify code. The contract remains intact.

This characteristic of containing data allows the user to created multiple instances of a given airplane without having to deal with dimensioned variables. In fact, by observing a rule forbidding data to exist outside of an object, all aircraft developed in the framework are multi-vehicle capable. If one wants multiple aircraft one just creates multiple instances of that aircraft.

Classes are the fundamental building blocks in C++. A class usually represents a single concept or physical entity. A developer specifies the behaviors (functions) and the data that a class will contain. An object is a particular instance of a class. It is useful and proper to think of classes like one thinks of intrinsic data types (int, float, double).

The two most common relationship between classes are inheritance and containment. Inheritance is often referred to as an "is a" relationship in that, when one class inherits from another there is an implicit assumption that the derived class is a either a specialization or an implementation of the base class. For example the B-757 "is a" Aircraft. Perhaps the most powerful use of inheritance is to provide abstract interfaces to subclasses. The base class can define access to behavior that is fully or completely defined in the derived classes. This mechanism, known as polymorphism, is implemented in C++ by use of virtual and pure virtual functions. The other common relationship is when a class contains a reference to a class or classes internal to itself. This is often the case when those classes are used to implement the containing class. This relationship is sometimes called "has a", containment, or aggregation.

The use of OO C++ and the enforcement of a few simple implementation guidelines within the framework create an environment where OO concepts are not just allowed, but fostered.

Framework Structure

The basic framework abstractions can be seen in Figure 1.

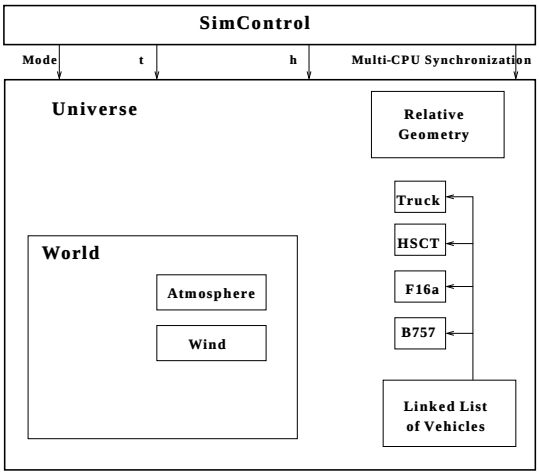

Figure 1: Framework Conceptual Diagram

All aircraft developed in the framework derive from a common set of abstract base classes that provide an interface between the framework and the specific aircraft being modeled (Figure 2).

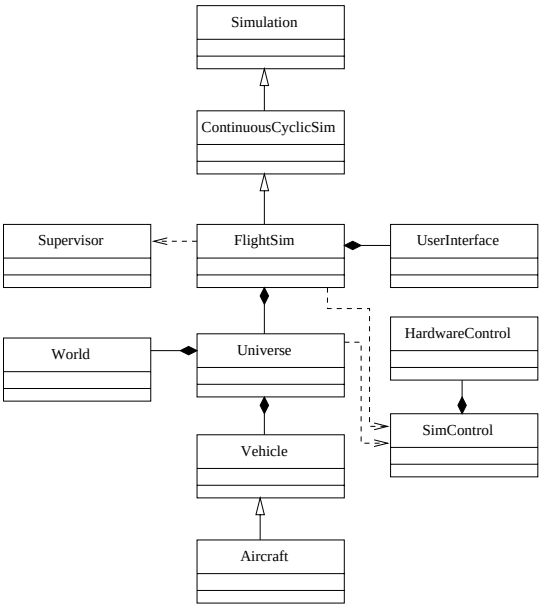

Figure 2: Top Level View Of Framework

A brief summary of the classes in Figure 2 follows:

- `Simulation`: Provides an abstract interface to allow concrete simulation types to be told to `execute()`.
- `ContinuousCyclicSim`: Adds an abstract interface that supports the concepts of mode dependent behavior and time that flows forward in fixed, discrete intervals (time step).
- `FlightSim`: Defines the initialization and execution behavior of dynamic vehicles.
- `Universe`: Provides an environment in which the vehicles execute. The concept of moving relative to a `World`, and relative geometry between vehicles are encapsulated in this class.
- `World`: Provides a world for the vehicles to fly around. `Wind` and `Atmosphere` are encapsulated here.
- `Vehicle`: An abstraction for a dynamic vehicle that has mode dependent behavior.
- `Aircraft`: Adds behavior specific to an airplane.
- `SimControl`: Provides moding and timing information in addition to multi-vehicle and multi-CPU synchronization.
- `HardwareControl`: An abstract interface to the hardware used in the simulation.
- `Supervisor`: Provides synchronization of simulation to hard real-time clock.

The modes used in the framework are defined as follows:

- reset: Used to initialize simulation/vehicles to a known state, time is set to zero.
- hold: Time does not increment, all states are frozen.
- operate: Time increments in discrete steps, vehicle dynamics are active.
- trim: Vehicles are driven to some defined steady state condition, time does not increment.

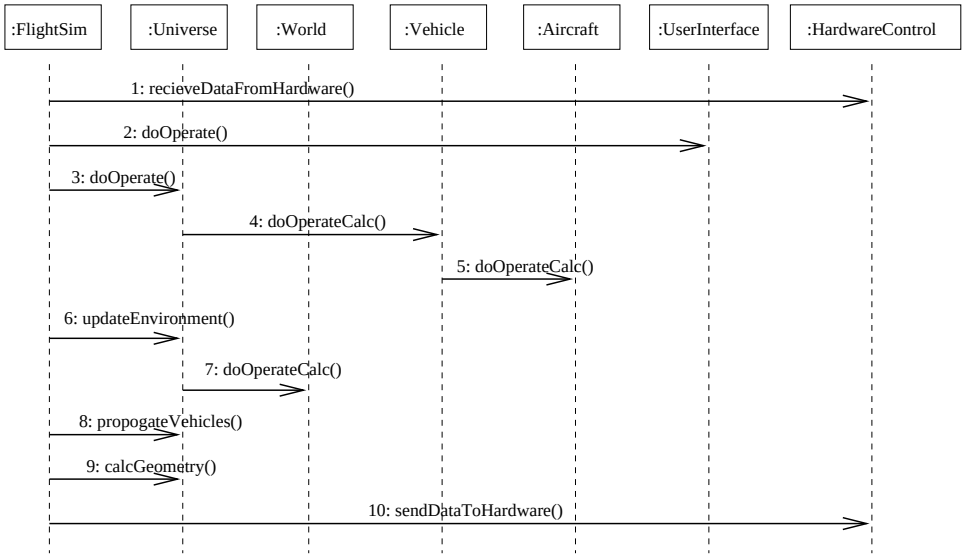

Figure 3: Top Level Object Interaction Diagram

At the highest level of abstraction the framework views all the the dynamic objects as vehicles, aircraft is just a specific kind of vehicle. Therefore, while primarily used to simulate aircraft, the framework is capable of supporting any type of dynamic vehicle. The framework provides real-time framing, mode (reset/hold/trim/operate) control, access to necessary variables, and the ability to do synchronous and asynchronous input/output to hardware. The classes describing a particular vehicle define the unique behavior that will be exhibited in a given mode.

For example, each vehicle defines its own unique behavior in operate mode(Figure 3). A brief description of each method follows:

- `HardwareControl::recieveDataFromHardware()`: Causes all synchronous and asynchronous data received from the various hardware components used in the simulation to be processed.
- `Universe::updateEnvironment()` Commands list of `World` objects to `doOperateCalc()`.
- `UserInterface::doOperate()`, `Universe::doOperate()`, `Vehicle::doOperate()`, `World::doOperateCalc()` Causes each concrete class associated with these abstract interfaces to perform whatever calculations they perform in operate.
- `Universe::propogateVehicles()` Causes the state to advance to the next time step.
- `Universe::calcGeometry()` Calculates relative geometry between vehicles.
- `HardwareControl::sendDataToHardware()` Sends the asynchronous and synchronous data generated by the simulation to hardware.

Since each vehicle defines its own behavior in the various modes (hold/trim/reset/operate), and each vehicle has slightly different components, the abstractions used are not necessarily the same across all vehicle types. However, it is useful to examine a typical abstraction for an airplane as shown in Figure 4. The interaction between the objects used to implement this airplane are shown in Figure 5.

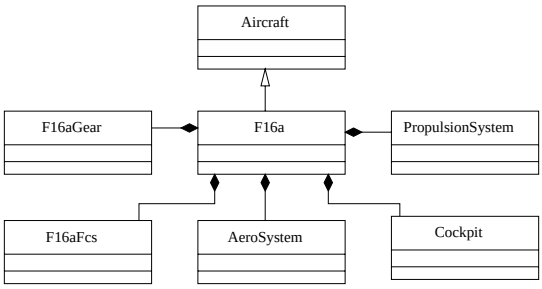

Figure 4: Typical Aircraft Architecture

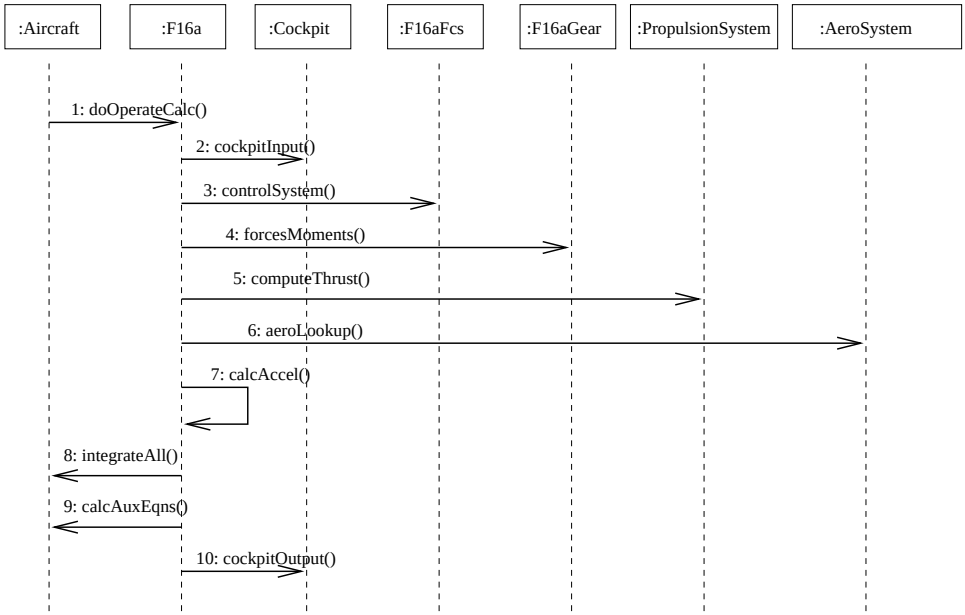

Figure 5: Typical Airplane Object Interaction Diagram

Interfaces to hardware are also abstract. Specific details of the interface to a particular piece of hardware are abstracted away and the user only has to deal with an interface that is generic. Cockpits are supported through an abstract interface that normalizes all inputs and outputs to the specific cockpit hardware.[1] While the full set of inputs needed to utilize all the capabilities of an airplane may not be present in a given cockpit, any airplane can be flown from any cockpit.

The framework is capable of supporting N-vehicles running on M-processors.[2] A current restriction is that a given vehicle's components must reside on the same CPU. Research is ongoing into the possibility of using a multi-threaded environment which would allow the distribution of a vehicle's components across CPUs.

The user interfaces with the simulation through a X-window Graphical User Interface (GUI) that provides the ability to create and delete aircraft from the simulation. Mode control, and the viewing and modification of variables in the simulation is also done through this GUI. The simulation can be operated from any X-based terminal. The same GUI interface that is used to operate production real-time is available to support checkout from the users desktop. A text based interface is also supported for use from ASCII terminals.

Due to the use of encapsulation in the simulation, the user can only modify variables through the use of methods that have been provided by the class designer in the public interface. The designer can choose to allow read-only, write-only, or read-write access to variables. Variables can also be completely encapsulated in the class and therefore protected from inappropriate modification.

In the process of developing the framework a set of "toolbox" classes were developed to provide developers with building blocks for the construction of simulation models. Toolbox functionality includes classes that:

- perform table lookups
- interface to SCRAMNet
- generate various random variate distributions
- provide both TCP/IP and TCP/UDP socket interfaces
- perform various filtering functions (such as Tustin's method)
- provide mutual exclusion.

To the extent possible the interfaces have been simplified so that users have access to capabilities that they might not have chosen to use in the past due to the complexity involved. The socket classes are an example of where users are now able to establish socket communications between distributed components in the simulations without having to deal with the details of how the interface is established.

Simulations Using Framework

To date there are four simulations supported by the framework. These simulations are a F-16A, a F-18E/F Drop Model,[3] a High Speed Civil Transport (AST-104) model and, a B-757. The B-757 is particularly interesting in that the code developed to drive experimental systems on the actual research B-757 are being developed in the simulation environment and will be moved to the aircraft with little (if any) modification. This concept called sim-to-flight has been facilitated by the use of object technology to develop the code.

Conclusions

A real-time simulation architecture has been developed utilizing OO analysis and design techniques. The architecture has been implemented using the OO programming language C++. This has resulted in a highly flexible system capable of simulating multiple instances of dynamic objects. These objects can be distributed across CPUs.

The software architecture is a self contained environment that is capable of supporting any type of dynamic object. All data is encapsulated in classes. There is no global or public data in this architecture.

Each dynamic object contains its own sequence of operations to execute. This allows each dynamic object to have uniquely defined behavior separate from other dynamic objects in the simulation. Multiple vehicles can be simulated simultaneously. The relative geometry of each vehicle relative to every other vehicle in the simulation is calculated.

The main routine driving this simulation is completely independent of both vehicle (plane, truck, ship, etc.) and simulation (continuous cyclic or discrete-event) type. This along with the supporting class library facilitates the rapid development of new types of vehicles through reuse.

Acknowledgments

The authors wish to thank the LaRC Simulation Systems Branch for their strong support in a time of change.

Bibliography

[1] P. Sean Kenney, et al. Using Abstraction To Isolate Hardware In An Object-Oriented Simulation. Paper Number AIAA-98-4533, August, 1998.

[2] Michael M. Madden, et al. Constructing A Multiple-Vehicle, Multiple-CPU Simulation Using Object-Oriented C++. Paper Number AIAA-98-4530, August, 1998.

[3] Kevin Cunningham, et al. Simulation Of A F/A-18 E/F Drop Model Using The LaSRS++ Framework. Paper Number AIAA-98-4160, August, 1998.

[4] Grady Booch. *Object-Oriented Analysis and Design*. Benjamin/Cummings, Redwood City, California, 1994.

[5] Joseph A. Kaplan, Patrick S. Kenney. SimGraph - A Flight Simulation Data Visualization Workstation. Paper Number AIAA-97-3797, August, 1997.

[6] David W. Geyer, et al. Managing Shared Memory Spaces In An Object-Oriented Real-Time Simulation. Paper Number AIAA-98-4532, August, 1998.

[7] Bruce Eckel. *Thinking in C++*. Prentice-Hall, Englewood Cliffs, New Jersey, 1995.

[8] Gamma E., Helm R., Johnson R., Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[9] John Lakos. *Large-Scale C++ Software Design*. Addison-Wesley, Reading, Massachusetts, 1996.

[10] Robert C. Martin. *Designing Object-Oriented C++ Applications Using The Booch Method*. Prentice-Hall, Englewood Cliffs, New Jersey, 1995.

[11] Scott Meyers. *Effective C++*. Addison-Wesley, Reading, Massachusetts, second edition, 1998.

[12] Scott Meyers. *More Effective C++*. Addison-Wesley, Reading, Massachusetts, 1996.

[13] David R. Musser, Atul Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, Reading, Massachusetts, 1996.

[14] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, Massachusetts, third edition, 1997.

[15] Patricia C. Glaab, et al. A Method To Interface Auto-Generated Code Into An Object-Oriented Simulation. Paper Number AIAA-98-4531, August, 1998.

[16] Terry Quatrani. *Visual Modeling With Rational Rose and UML*. Addison Wesley, Reading, Massachusetts, 1998.